EL LLENGUATGE ERLANG

PROGRAMACIÓ CONCURRENT I EN TEMPS REAL

Antoni Escobet

Manresa, setembre de 2022

Índex

1. Introducció a Erlang
2. Programació concurrent
3. Nodes i sistemes distribuïts

Enllaços

- Instal·lació
  - http://www.erlang.org/downloads
- Documentació
  - http://erlang.org/doc/index.html
  - https://es.wikibooks.org/wiki/Programación_en_Erlang
  - http://aprendiendo-erlang.blogspot.com.es/
- Llibres
  - http://erlang-otp.es/

- https://www.netguru.com/blog/10-companies-use-elixir
- https://www.locurainformaticadigital.com/2019/04/23/stackoverflow-lenguajes-programacion-mejor-pagados/

Índex

1. **Introducció a Erlang**
2. Programació concurrent
3. Sistemes distribuïts

El llenguatge Erlang

- Joe Armstrong. 1986 als laboratoris Ericsson
- Va ser un llenguatge propietari fins el 1998
- Objectiu inicial: Programar aplicacions telefòniques
- Llenguatge funcional amb una sintaxi heretada de Prolog, tolerant a fallades i orientat a les tasques de temps real i concurrència.
- Erlang és un entorn de programació
  - Està format per un compilador de codi, una col·lecció d'eines i una màquina virtual
- Punts de vista
  - Erlang com a llenguatge: Hibrid (funcional, imperatiu i una mica de programació orientada a objectes). És un llenguatge orientat a la concurrència.
  - Erlang com entorn d'execució: Primer és pseudocompila (compila de *.erl a *.beam) i després s'executa sobre una màquina virtual

El llenguatge Erlang

Llenguatges que utilitzen la màquina virtual d'Erlang:

- Reia http://reia-lang.org/
- Efene http://efene.org/
- Joxa http://joxa.org/
- LFE http://lfe.io/
- ELIXIR https://elixir-lang.org/

El llenguatge Erlang: Característiques

Distribuït: Per poder repartir la càrrega en diferents màquines.

Tolerant a fallades: Una errada a una part dels sistema no pot aturar-ho tot.

Escalable: Ha de poder gestionar milions de processos.

Modificar el codi en calent: No s'ha d'aturar el sistema per fer una actualització.

El llenguatge Erlang: Característiques

Assignacions úniques:

- El valor a una variable s'assigna un sol cop.

Llenguatge simple:

- Pocs elements
- Cap excepció
- Només dues estructures de control: "case" i "if"
- No hi ha bucles
- Utilitza tècniques de recursivitat i modularització

Orientat a la concurrència:

- Les rutines internes faciliten la realització de programes concurrents i distribuïts.

Pas de missatges:

- La comunicació entre els processos es realitza només amb missatges

El llenguatge Erlang: Desenvolupaments

Sector empresarial:

- **Facebook** utilitza Erlang en el “chat”
- **Tuenti**
- **Demonware** per suportar els usuaris de videojocs com “Call of Duty”
- **Wooga** per aplicacions de mòbils
- **WhatsApp** l'utilitza a nivell de servidor per gestionar els missatges

Programari lliure

Intèrpret de comandes

- erl és el intèrpret de comandes o màquina virtual
  1. $ erl

     Erlang/OTP 21 [erts-10.0] [64-bit] [smp:8:8] [async-threads:10]

     Eshell V10.0 (abort with ^G)

     1>
- El intèrpret s'atura amb ^G (control-G) o amb q().
  - Per sortir s'ha de prémer q. Hi ha més opcions de depuració
- $ erl –man lists → Mostra ajuda sobre els mòduls (Unix)

Tipus de dades bàsics

Sencers, reals, cadenes de caràcters, etc.

Sencers

> 5.

1. > 2 + 3.

   5

2. > 123456789 * 56789012 * 998989281928192 * 877676767.

   614716288757274939816291044291021327155

Cadenes de caràcters

3. > “abc”.

   “abc”

Reals

4. > 2.0.

   2.0

5. > 2.7/6.

   0.45

Tipus de dades

Àtoms

- Son identificadors de tipus caràcter que s'utilitzen com a paraules clau. És una paraula que comença per una lletra en minúscules. També pot començar amb majúscules, nombres, espais, ... però s'ha de posar entre cometes simples.

```
Ret = fitxer_operacio(op_lect, Buff),
If
   Ret == 'Ret_succés' ->
                 ....
```

  - op_lect i 'Ret_succés' son àtoms
  - Es poden comparar
  - "," s'utilitza per encadenar sentències

La seva finalitat és només ajudar al programador a identificar estructures, algoritmes i codi concret.

Tipus de dades bàsics

Variables

- Només és pot assignar un sol valor, i és invariable.
- Format: Lletra majúscula al primer caràcter.

```
8> X = 12345.          % S'assigna un valor
   12345
9> X.
   12345
10> X = 54321.         % Error
   ** exception error: no match of right hand side value 54321
11> f().               % Esborrar les variables
   ok
12> X = 54321.
   54321
```

L'assignació funciona com a les matemàtiques. Les dues parts han de ser certes.

```
1> X = 12345.
   12345
2> X = 12345.
   12345
3> Y = X + 1.
   12346
```

Tuples

- Les tuples son col·leccions de valors entre claus "{ }"

```
4> P = {100, 200}.
{100,200}
```

- Els àtoms es poden utilitzar per etiquetar camps

```
9> Persona = {persona,
9> {nom, pau},
9> {alçada, 1.84},
9> {n_peu, 43},
9> {color_ulls, marro}}.
.....
```

```
{persona,{nom,pau},
        {alçada,1.84},
        {n_peu,43},
        {color_ulls,marro}}
```

- Extracció de valors

```
10> {persona, {nom, N}, {alçada, H}, _ , _} = Persona.
.....
11> N.
pau
12> H.
1.84
```

Llistes

- Les llistes son vectors d'informació, que pot ser de diferents tipus (nombres, àtoms, tuples, llistes).
- Les llistes es creen amb claudàtors "[ ]":

```
13> Y = [1,2,3,4].
[1,2,3,4]
14> CosesAComprar = [{pomes,8}, {peres,4}].
[{pomes,8},{peres,4}]
```

- Les llistes es poden construir amb l'operador "|":

```
1> [H|T] = [1,2,3,4].
[1,2,3,4]
2> H.
1
3> T.
[2,3,4]
```

```
5> [{Item, Nombre}|RestaCosesComprar] = CosesAComprar.
[{pomes,8},{peres,4}]
6> Item.
pomes
7> Nombre.
8
```

Llistes (Operadors)

- ++ i --
  - A ++ B: Afegeix a la llista A la llista B.
  - A -- B: Extreu de la llista A la llista B.
  - Exemples:
    - [1,2,3] ++ [4,5,6]. →[1,2,3,4,5,6]
    - [a,b,c,1,d,e,1,x,y,1] -- [1]. → [a,b,c,d,e,1,x,y,1]
    - [a,b,c,1,d,e,1,x,y,1] -- [1,1,1]. → [a,b,c,d,e,x,y]
    - [a,b,c,1,d,e,1,x,y,1] -- [1,1,1,1]. → [a,b,c,d,e,x,y]
- ++ en patrons
  - ++ es pot utilitzar en els patrons. En la relació de cadenes, podem escriure patrons:
    - f("inici" ++ T) -> ...
    - f("final" ++ T) -> ...

Una paraula entre cometes dobles representa una cadena de text, que és un tipus concret de llista

Cadenes de caràcters

- Les cadenes de caràcters son una sèrie de nombres:

```
1>  [83, 111, 114, 112, 114, 101, 115, 97].
"Sorpresa“
```

Tots els caràcters imprimibles

```
2> [1, 34, 67, 78, 98, 101, 123, 145, 167, 201].
[1,34,67,78,98,101,123,145,167,201]
```

Amb caràcters no imprimibles

- Els caràcters es poden aconseguir amb “$”:

```
3> [$s, $i].
"si"
```

Llistes binàries

- Les llistes binàries son per emmagatzemar cadenes de caràcters d’un byte i es poden realitzar tasques específiques amb seqüències de bytes o de bit.

  1> << “Hola“ >>

  <<“Hola“>>

  2> <<“ 72, 111, $l, $a”>>.

  <<“Hola“>>

- No es poden afegir i extreure elements
- Per concatenar llistes binàries:

  1> A = << “Bon” >>. → <<“Bon”>>

  2> B = << “dia” >>. → <<“ dia”>>

  3> C = << A/binary, B/binari” >>. → <<“Bon dia”>>

Altres tipus de dades

- **Llistes de propietats:** És una llista de tuples {clau, valor}. Es gestiona amb la llibreria **proplists.**
  - > A = [{dia, “10”}, {mes, “Gener”}, {any, “2016”}].
  - proplists:get_value(mes, A). → “Gener”

- **Registres:** És un tipus específic de tuples. Es una estructura per emmagatzemar un nombre fixe d’elements. (similar a una estructura de C)

Mòduls

Fitxer figuraGeo.erl:

```
-module(figuraGeo).
-export([area/1]).
area({rectangle, Amplada, Ht}) -> Amplada * Ht;
area({cercle, R}) -> 3.14159 * R * R.
```

- Tot codi executable s'especifica en un mòdul.
- El nom del mòdul ha de ser el mateix que el nom del fitxer .erl
- S'ha de compilar i es genera un fitxer .beam.
- Es carrega automàticament accedint al seu espai de noms "mòdul:funció_exportada".
- A les funcions s'afegeix el nombre de paràmetres (area/1).
- L'especificació parcial es separa per: ";".
- Només s'exporten les funcions especificades en el "export".

Mòduls

- Compilació:

```
8> c(figuraGeo).
{ok,figuraGeo}
```

- Ús:

```
10> figuraGeo:area({rectangle, 10, 3}).
30
11> figuraGeo:area({cercle, 1.5}).
7.068577499999999
```

- Observacions:
  - Es composa incrementalment (open/closed modules)
    - Es podria afegir una altre funció:

```
area({quadrat, X}) -> X * X;
```

Mòduls

- Llista de la compra:

  1> LlistaCompra = [{ taronges ,4}, { diaris,1}, {pomes ,10}, {préssecs ,6}, {llet,3}].

  [{taronges,4},{diaris,1},{pomes,10},{préssecs,6},{llet,3}]

- Botiga.erl

```
-module (botiga).
-export ([cost/1]).
cost(taronges ) -> 5;
cost(diaris ) -> 8;
cost(pomes ) -> 2;
cost(préssecs ) -> 9;
cost(llet) -> 7.
```

Es pot afegir una funció que calculi el cost de la compra

```
-module (botiga).
-export ([cost/1, total/1]).
cost(taronges ) -> 5;
cost(diaris ) -> 8;
cost(pomes ) -> 2;
cost(préssecs ) -> 9;
cost(llet) -> 7.

total([{Que, N}|T]) -> botiga:cost(Que) * N + total(T);
total([]) -> 0.
```

Mòduls

- Ús:

1> LlistaCompra = [{ taronges ,4}, { diaris,1}, {pomes ,10}, {préssecs ,6}, {llet,3}].

[{taronges,4},{diaris,1},{pomes,10},{préssecs,6},{llet,3}]

2> c(botiga).

{ok,botiga}

Compilació

pattern machine

3> botiga:total([{llet, 5}, {pomes,10}]).

55

Llista de tuples

Cost de la llista de la compra:

- 5 ampolles de llet
- 10 pomes

4> botiga:total(LlistaCompra).

123

Funcions i acumuladors

15/9

Sumar els elements d'una llista

```
sum (L) -> sum(L, 0).

sum ([], N) -> N;
sum ([H|T], N) -> sum(T, H+N).
```

```
sum([H|T]) -> H + sum(T);
sum([]) -> 0.

Ex: sum([3,2]) ->
          3 + sum([2])
          2 + sum([])
          0
```

```
Ús:     c(sum).
        {ok,sum}
        1> sum:sum([3]).      3
        2> sum:sum([3,2,3,5,10],17).  40
```

- Versions amb un i dos paràmetres (sum/1 i sum/2)
- L'acumulador s'utilitza per acumular el resultat
- Propietat fonamental → *Tail Call Optimization (TCO)*
  - La crida recursiva apareix l'última en el cos de la funció
  - A la crida s'elimina el context de pila (es converteix en un bucle)

Funcions anònimes

22/9

En altres llenguatges se'n diuen: funcions.
Cada *fun* té associada una referència.

```
1>Z = fun(X) -> X+2 end.
#Fun<erl_eval.6.82930912>
2>Z(4).
6
3>TemperaturaC = fun({c,C}) -> {f, 32 + C*9/5};
3>({f,F}) -> {c, (F-32)*5/9} end.
#Fun<erl_eval.6.82930912>
4>TemperaturaC({c,100}).
{f,212.0}
5>TemperaturaC({f,212}).
{c,100.0}
6>TemperaturaC({c,0}).
{f,32.0}
```

```
7> TemperaturaC({e,100}).
** exception error: no function clause matching
        erl_eval:'-inside-an-interpreted-fun-'({e,100})
```

Funcions anònimes


```
1> EsParell = fun(X) -> (X rem 2) =:= 0 end.
#Fun<erl_eval.6.82930912>
2> EsParell(8).
true
3> EsParell(7).
False
4> lists:map(EsParell, [1,2,3,4,5,6,7,8]).
[false,true,false,true,false,true,false,true]
5> lists:filter(EsParell, [1,2,3,4,5,6,7,8]).
[2,4,6,8]
```

llibreria de funcions

Mòdul lists (Per la gestió de llistes)

all(Pred, List)

any(Pred, List)

dropwhile(Pred, List)

filter(Pred, List)

foldl(Fun, Acc0, List) -> Acc1

foldr(Fun, Acc0, List) -> Acc1

map(Fun, List1) -> List2

partition(Pred, List) -> {Satisfying, NotSatysfying}

Funció “map”


```
-module (mp).
-export ([map /2, map /3]).

map (_, []) -> [];
map (F, [H|T]) -> [F(H)| map(F, T)].

map (_, [], R) -> lists : reverse (R);
map (F, [H|T], R) -> map(F, T, [F(H)|R]).
```

```
1> c(mp).
{ok,mp}
2> mp:map(EsParell , [1 ,2 ,3 ,4 ,5 ,6 ,8]).
[false,true,false,true,false,true,true]
3> lists:map(EsParell , [1 ,2 ,3 ,4 ,5 ,6 ,8]).
[false,true,false,true,false,true,true]
```

- F → funció
- [L] → llista

Agafa el primer element de la llista L i l’aplica a la funció F

- map/2 també es podia haver implementa en base a map/3:

  map(F, L) -> map(F, L, []).
- map/3 també es podia haver implementat amb l’operador de concatenació de llistes “++” (Exercici)

Funcions anònimes (tancament)


```
1> Mult = fun(Temps) -> (fun(X) -> X * Temps end ) end.
#Fun<erl_eval.6.82930912>
2> Triple = Mult(3).
#Fun<erl_eval.6.82930912>
3> Triple(4).
12
```

- Una funció anònima pot incloure paràmetres preestablerts.
- A l'exemple Temps queda lligat dins de la funció.
- La funció més el conjunt de lligadures → tancament

```
-module(misc).
-export([for/3]).

for(Max, Max, F) -> [F(Max)];
for(I, Max, F) -> [F(I) | for(I+1, Max, F)].
```

Condició de finalització

Cas recursiu

```
1> c(misc).
{ok,misc}
2> misc:for(0,10,Triple).
[0,3,6,9,12,15,18,21,24,27,30]
```

Llistes de comprensió (Lists Comprehensions)

- Les llistes de comprensió, són expressions que creen llistes sense haver d'usar funcions anònimes, mapes, o filtres. Això fa que els nostres programes siguin encara més curts i fàcils d'entendre.
- Les llistes de comprensió consisteix en una representació matemàtica de conjunts. Per exemple, P = {x ∈ N / x és menor de 5}, expressa el conjunt dels x tals que x és un nombre natural menor de 5.

```
1> L = [1,2,3,4,5].
[1,2,3,4,5]
2> [2*X || X <- L].
[2,4,6,8,10]
3> Compra = [{taronges, 4}, {diaris, 1}, {pomes, 10}, {préssecs, 6}, {llet, 3}].
[{taronges,4},{diaris,1},{pomes,10},{préssecs,6},{llet,3}]
4> [{Nom, 2*Nombre} || {Nom, Nombre} <- Compra].
[{taronges,8},{diaris,2},{pomes,20},{préssecs,12},{llet,6}]
```

Exemple

```
1> [X || X <- [1,2,3,4,5,6,7,8,3,2], X < 5].
[1,2,3,4,3,2]
```

- Pregunta: Com es faria *map* amb llistes de comprensió?

Llistes de comprensió. Exemple

- Mètode d'ordenació: Quicksort

```
qsort ([]) -> [];
qsort ([ Pivot |T]) ->
        qsort ([X || X <- T, X < Pivot ])++ [ Pivot ] ++ qsort ([X || X <- T, X >= Pivot ]).
```

```
1> c(oper).
{ok,oper}
2> oper:qsort([12,2,23,1,14,3,9,45]).
[1,2,3,9,12,14,23,45]
```

Exemple: oper:qsort([12,2,23,1,14,3,9,45]).

Particions (llista / pivot)	Resultat
[12,2,23,1,14,3,9,45] 12	[1,2,3,9,12,14,23,45]
[2,1,3,9] 2 ; [23,14,45] 23	[1,2,3,9][14,23,45]
[1] 1 ; [3,9] 3 ; [14] 14 ; [45] 45	[1,3,14] [45]
[9] 9	[9]

Llistes de comprensió. Exemple

- Triplets de Pitàgores: $A^2 + B^2 = C^2$:

```
pythag (N) ->
            L = lists:seq(1,N),
            [ {A,B,C} ||
                        A <- L,
                        B <- L,
                        C <- L,
                        A+B+C =< N,
                        A*A+B*B =:= C*C
            ].
```

```
1> c(oper).
{ok,oper}
2> oper:pythag(9).
[]
3> oper:pythag(12).
[{3,4,5},{4,3,5}]
4> oper:pythag(24).
[{3,4,5},{4,3,5},{6,8,10},{8,6,10}]
```

Es genera una llistes L amb una seqüència d'enters des d'1 a N per generar els valors de A, B i C. A la llista de comprensió hi ha tres generadors, un per a cada valor del triplet amb valors compresos entre 1 i N i a continuació, les condicions del problema.

Llistes de comprensió. Exemple

- Permutacions:

```
perms([]) -> [[]];
perms(L) -> [[H|T] || H <- L, T <- perms(L -- [H])].
```

La idea és ben simple: prenem un element de la llista i permutem la resta. Així, tenim totes les permutacions del primer element, les del segon element, etc ...
L'operador "—" fa la diferència entre dues llistes, és a dir, elimina de la primera llista dels elements de la segona.

```
1> c(oper).
{ok,oper}
2> oper:perms("123").
["123","132","213","231","312","321"]
139> oper:perms("gat").
["gat","gta","agt","atg","tga","tag"]
```

Per calcular totes les permutacions de X12, calcular totes les permutacions de 12:

[12 21]

Ara intercalar la X en totes les posicions possibles en cada permutació, de manera que al afegir X a 12 dóna:

X12 1X2 12X,

afegint X a 21 dóna:

X21 2X1 21X,

i així successivament.
Apliqueu aquestes regles de forma recursiva.

Guardes

- Les guardes són una construcció booleana que ens permet o facilita la selecció d'una clàusula concreta d'un conjunt d'elles. La sintaxi d'una funció amb guarda és nom_clàusula (paràmetres) *when* guarda ->
- Suposem que volem fer una funció que ens proporcioni el màxim de dos nombres.

```
max ( A, B ) when A > B ->
    A;
max ( _, B ) ->
    B.
```

La paraula màgica *when* indica que la primera clàusula s'executi quan es compleixi la condició. En aquest cas, quan A sigui més gran que B.

Una seqüència de guardes és un conjunt de guardes separades per comes o punts i comes.

when G1; G2; ...; GN ->: La clàusula s'executarà quan algunes de les guardes sigui certa.
when G1, G2, ..., GN ->: La clàusula s'executarà quan totes les guardes siguin certes.

Guardes

- Fins ara hem dit que les guardes són sentències booleanes. Però no totes les expressions booleanes són vàlides per a l'ús de guardes.
- Sentències vàlides per guardes:

> - L'àtom cert
> - Altres constants i variables que avaluen a un booleà
> - Crides a funcions BIF (funcions especials d'Erlang)
> - Comparacions
> - Expressions booleana (not, and, or i xor)
> - Expressions booleanes de curt circuits (orelse i andalso)

Guardes

Declaracions

Funcions Bif	Significat
is_atom(X)	X és un àtom?
is_binary(X)	X és un binari?
is_constant(X)	X és una constant?
is_float(X)	X és un real?
is_function(X)	X és una funció?
is_function(X, N)	X és una funció amb N arguments?
is_integer(X)	X és un sencer?
is_list(X)	X és una llista?
is_number(X)	X és un sencer o un real?
is_pid(X)	X és un identificador d'un procés?
is_port(X)	X és un port?
is_reference(X)	X és una referència?
is_tuple(X)	X és una tupla?
is_record(X,Tag)	X és un registre del tipus Tag?
is_record(X,Tag,N)	X és un registre del tipus Tag i de N elements?

Guardes

Funcions

Funció	Significat
abs(X)	Valor absolut de X.
element(N, X)	Element N de X. Nota X és una tupla.
float(X)	Converteix X (nombre) a un real.
hd(X)	El primer element de la llista X.
length(X)	La longitud de la llista X.
node()	El node actual.
node(X)	El node en què X s'ha creat. X pot ser un procés. Un identificador, una referència o un port.
round(X)	Converteix X (nombre) a un sencer.
self()	L'identificador de procés del procés actual.
size(X)	La mida de X. X pot ser una tupla o un binari.
trunc(X)	Trunca X (nombre) a un nombre sencer.
tl(X)	L'últim element de la llista X.

Guardes Exemples


```
f(X,Y) when is_integer(X), X > Y, Y < 6 -> ...
```

Quan X és un nombre enter i X és més gran que Y i Y és menor que 6." La coma, que separa les guardes, significa "i"

```
when

is_tuple(T), size(T) =:= 6, abs(element(3, T)) > 5
element(4, X) =:= hd(L)
```

hd -> Cap de la llista
td -> Cua de la llista

La primera línia significa que T és una tupla de sis elements i el valor absolut del tercer element de T és mes gran que 5. La segona línia vol dir que l'element 4 de la tupla X és idèntica al primer de la llista L.

```
when

X =:= gat; X =:= gos
is_integer(X), X > Y ; abs(Y) < 23
```

La primera guarda diu que X és un gat o un gos. La segona guarda vol dir que X és un nombre sencer, que X és més gran que Y o que el valor absolut de Y és menor que 23.

Guardes

Exemples

Guardes amb curtcircuit booleana:

```
A >= -1.0 andalso A+1 > B
is_atom(L) orelse (is_list(L) andalso length(L) > 2)
```

La raó per permetre expressions booleanes a les guardes és per fer que les guardes siguin sintàcticament similar a altres expressions. La raó dels operadors *orelse* i *andalso* és que els operadors booleans i/o van ser definits originalment per avaluar tots els seus arguments. A les guardes, pot haver diferències entre (*and* i *andalso*) o entre (*or* i *orelse*).

```
f(X) when (X == 0) or (1/X > 2) ->
...
g(X) when (X == 0) orelse (1/X > 2) ->
...
```

La guarda de f(X) dona un error de divisió per quan X val zero.
La guarda de g(X), quan X = 0 funciona correctament, ja que no analitza la segona expressió.
A la pràctica, pocs programes usen guardes complexes, amb guardes simples (,) sol ser suficient per a la majoria dels programes.

Registres (records)

Els registres permeten definir estructures de dades complexes amb nom:

```
-record (Nom , {
                    %% les següents claus tenen un valor per defecte
          clau1 = Default1 ,
          clau2 = Default2 ,
...
                    %% Clau sense valor per defecte : %% key3 = undefined
          clau3 ,
...
}).
```

- Es defineix en un fitxer .hrl
- Al intèrpret de comandes s'utilitza rr ("fitxer.hrl") per llegir les definicions dels registres

Registres


```
1> rr("Registre1.hrl").
[tot]
2> X = #tot{}.
#tot{estat = avis, qui = joan, text = undefined}
3> X1 = #tot{estat = urgent, qui = joan, text = "Arreglar una errada"}.
#tot{estat = urgent, qui = joan, text = "Arreglar una errada"}
4> X2 = X1#tot{estat = fet}.
#tot{estat = fet, qui = joan, text = "Arreglar una errada"}
```

Fitxer: Registre1.hrl

```
-record (tot, {estat = guardant, qui = joan, text}).
```

A les línies 2 i 3 es creen nous registres. La sintaxi #tot{clau1=val1, ..., clauN=valn} s'utilitza per crear un nou registre de tipus *tot*.
Les claus són tots els àtoms i ha de ser igual als utilitzats en la definició de registre.
Si una clau s'omet, llavors s'assumeix un valor per defecte que ve de la definició del registre.

A la línia 4 copiem un registre que ja existeix. La sintaxi *X1#tot{estat=fet}* significa crear una còpia de *X1* (que ha de ser de tipus *tot*), canviant el valor del camp *estat* a *fet*.
Aquesta és una còpia del registre original, el registre original no es modifica.

Extracció de valors:

```
5> #tot{qui = W, text = T} = X2.
#tot{estat = fet,qui = joan,text = "Arreglar una errada"}
6> W.
joan
7> T.
"Arreglar una errada"
```

Accés:

```
8> X2#tot.text.
"Arreglar una errada"
```

Registres:

Podem escriure funcions de pattern machine per modificar els camps d'un registre i crear nous registres.

```
clear_status(#tot{estat=S, qui=W} = R) ->
%%
        R#tot{estat=fet}
%% ...
```

Perquè coincideixi amb un registre d'un tipus particular, podem escriure la definició de la funció:

```
do_something(X) when is_record(X, tot) ->
%% ...
```

Aquesta guarda es certa quan X és un registre de tipus “tot”.

Els registres només són tuples. Per eliminar la definició de tot del shell:

```
9> X2.
#tot{estat = fet, qui = joan, text = "Arreglar una errada"}
10> rf(tot).
ok
11> X2.
{tot, fet, joan, ”Arreglar una errada"}
```

Elimina la definició del registre de *tot*.

X2 es mostra com una tupla

Els registres són una comoditat sintàctica per referenciar els diferents elements en una tupla.

Sentència: Case


```
case Expressió of
     Patro1 [when GuardSeq1] -> seqüència de comandes 1;
     ....;
     PatroN [when GuardSeqN] -> Seqüència de comandes N
  end
```

```
cas(A) ->
     case A of
          0 -> io:format("Val 0~n");
          1 -> io:format("No val 0~n")
end.
```

```
filtre(P,[H|T]) ->
          case P(H) of
                    true  -> [H|filtre(P, T)];
                    false -> filtre(P, T)
          end;
filtre(P, []) ->
          [].
```

La sentència *case* avalua l'expressió i comprova la coincidència amb el patró1 (condicionat a una guarda) i si es produeix la coincidència executa la seqüència de comandaments corresponent i en el cas que no hi hagi coincidència passa al següent patró.

```
filtre(P, [H|T]) -> filtre1(P(H), H, P, T);

filtre(P, []) -> [].

filtre1(true, H, P, T) -> [H|filtre(P, T)];

filtre1(false, H, P, T) -> filtre(P, T).
```

```
1> c(oper).
{ok,oper}
2> oper:filtre(fun(X) -> X rem 2 =:= 0 end, [1,2,3,4]).
[2,4]
```

Sentència: If


```
if
   Condicio1 ->
      Seqüència de comandes 1;
   ...;
   CondicioN ->
      Seqüència de comandes N
 end
```

Primer avalua la primera condició i si és certa a continuació avalua les expressions o comandaments corresponents a la condició 1.
Si la primera condició no és certa avalua la següent condició i així successivament fins que una té com a resultat 'cert'.

```
comparar(X,Y) ->
        if
                X>Y      -> major;
                X =:= Y  -> igual;
                true     -> menor
        end.
```

```
1> c(oper).
{ok,oper}
2> oper:comparar(3,4).
menor
```

Acumuladors

- "Recorden" resultats intermedis

- Com podem fer dues llistes d'una funció?
- Com podem escriure una funció que divideixi una llista d'enters en dues llistes que contenen els enters parells i imparells de la llista original?

```
parell_i_senar (L) ->
        Senar = [X || X <- L, (X rem 2) =:= 1],
        Parell = [X || X <- L, (X rem 2) =:= 0],
        {Senar , Parell}.
```

```
1> c(oper).
{ok,oper}
2> oper:parell_i_senar([1 ,2 ,3 ,4 ,5 ,6]).
{[1,3,5],[2,4,6]}
```

Fa el recorregut de la llista dues vegades

Com es pot evitar?

Acumuladors


```
parell_i_senar([H|T], Senar, Parell) ->
        case (H rem 2) of
                0 -> parell_i_senar(T, Senar, [H|Parell]);
                1 -> parell_i_senar(T, [H|Senar], Parell)
        end;
parell_i_senar([], Senar, Parell) -> {Senar , Parell}.
parell_i_senar(L) -> parell_i_senar(L, [], []).
```

```
1> c(oper).
{ok,oper}
2> oper:parell_i_senar2([1, 2, 3, 4, 5, 6]).
{[5,3,1],[6,4,2]}
3> oper:parell_i_senar2(lists:reverse([1, 2, 3, 4, 5, 6])).
{[1,3,5],[2,4,6]}
```

Fa el recorregut de la llista un sol cop

Si es vol retornar la llista en el mateix ordre → lists:reverse(X)

Excepcions


```
funcio1(X) ->
    {P, Q} = funcio2(X),
    io:format("Q = ~p~n" , [Q]),
    P.
```

En aquest cas, quan eliminem la línia "io" donarà un missatge d'error dient que la variable "Q" no s'utilitza. Per evitar-ho podem escriure el mateix codi de la següent forma:

```
funcio1(X) ->
    {P, _Q} = funcio2(X),
    io:format("Q = ~p~n" , [_Q]),
    P.
```

Comandes de la consola

Mòduls

- c('Nom') : Compila el fitxer Nom
- l(M): Càrrega del mòdul M
- m(): Mostra tots els mòduls carregats a memòria
- m(M): Mostra informació detallada del mòdul M.
- lc([F]): Llista de fitxers a compilar
- nl(M): Carrega el mòdul M a tots els nodes connectats
- nc('Nom'): Compila i carrega el mòdul o fitxer a tots els nodes connectats

Variables

- b(): Mostra totes les variables
- f(): Elimina totes les variables de la consola

Històric

- h(): Mostra l'històric de comandes executades
- e(N): Repeteix la comanda N
- v(N): Mostra la línia de comanda N de l'històric

Comandes de la consola

Processos

- bt(Pid) : Obté el traçat de la pila del procés Pid
- flush(): Mostra tots els missatges enviats a la consola
- pid(X,Y,Z): Obté el tipus de dades Pid
- regs(): Mostra els processos registrats
- i(): Mostra tots els processos del node
- ni(): Mostra tots els processos de tots els nodes

Sortir

- q(): Sortir de la consola

Seguiment dels programes

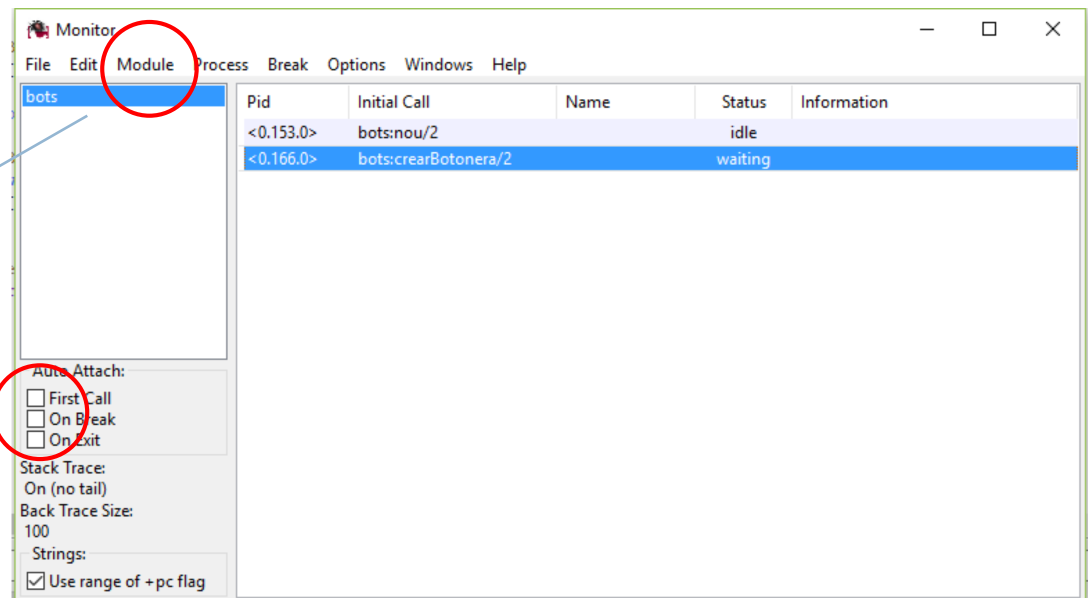
Pot ser l’eina més important i que millor s'hauria d'aprendre a utilitzar per poder analitzar el codi realitzat d'una forma més propera a la informació.

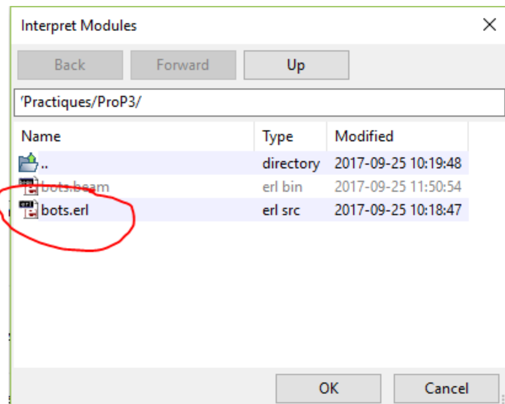
S’ha de compilar el mòdul activant la opció: debug_info

> c(‘Nom del mòdul’, debug_info).

El depurador s’obre amb la comanda:

- debugger:start().
- Module > Interpret





Errors d'execució

- **Function_clause**: Paràmetres incorrectes d'una funció, (per nombre de paràmetres, concordança o per guardes)
- **Case_clause**: Similar a l'anterior. S'activa quan no hi ha concordança amb cap bloc (o amb les guardes, si n'hi ha) dins d'un *case*
- **If_clause**: S'activa quan no hi ha cap guarda del *if* que es pugui aplicar
- **Badmatch**: Quan falla la concordança (matching), al intentar assignar una estructura de dades sobre una altre que no té la mateixa forma o quan s'intenta fer una assignació sobre una variable que ja té un valor
- **Badarg**: S'activa quan cridem una funció amb arguments erronis.
- **Undef**: Quan es crida a una funció que no existeix. Tan pot ser pel nombre de paràmetres com pel nom del mòdul
- **Badarith**: Excepció donada per errors matemàtics. Una operació amb valors erronis, una divisió per zero, ..
- **Badfun**: Quan es vol utilitzar una variables que no conté una funció.
- **Badarity**: És una cas concret de badfun, i es dona quan el nombre d'arguments de la funció ni és correcte.
- **System_limit**: S'ha arribat al límit del sistema. Hi ha masses processos, o masses arguments en una funció, o àtoms massa grans, o masses nodes, ...

Índex

1. Introducció a Erlang
2. **Programació concurrent**
3. Sistemes distribuïts

Processos

- Són ràpids de crear i destruir.
- L'enviament de missatges entre processos és molt ràpid.
- És fàcil mantenir un gran nombre de processos.
- Són molt lleugers.
- Són independents i no comparteixen memòria.
- Un procés compleix els principis del ser viu:
  - Neix (es crea)
  - Creix (ampliant els seus recursos assignats)
  - Reproduir-se (Genera altres processos)
  - Morir (finalitzar l’execució)

Primitives bàsiques: Generar, enviar i rebre

- **Pid = spawn(Fun)**

Crea un nou procés concurrent on *Fun* és la funció concurrent. El nou procés pot ser avaluat en paral·lel. La funció *spawn* retorna el *Pid* del procés que podrem utilitzar per a l'enviament de missatges al procés. El *Pid* del procés actual s'obté amb la funció self().
Pid => <X,Y,Z>   X : node (el terminal és el zero); Un procés s'identifica amb 18 bits
Y : Primers 15 bits; Z : Els bits del 16 al 18

- **Pid! Missatge**

Aquesta construcció ens permet l'enviament de missatges al procés identificat pel *Pid*. El missatge és asíncron, per la qual cosa, no hem d'esperar que el procés ens respongui. Quan fem *Pid! Missatge* Erlang ens respon sempre amb *Missatge*. Es pot enviar el mateix missatge a diferents processos, per exemples, Pid1! Pid2! Pid3! Missatge s'envia el missatge a tots els processos en l'ordre establert.

Processos

• **receive ... End**

La construcció *receive* s'encarrega de rebre i tractar el missatge enviat al procés.

```
receive
        Patro1 [when Guarda1] -> seqüència de comandes 1;
        ...
        PatroN [when GuardaN] -> seqüència de comandes N
end
```

- Quan un missatge arriba al procés, el sistema intenta aparellar-se amb el *Patro1* (amb una possible guarda *Guarda1*), si té èxit, s'avalua la *seqüència de comandes 1*. Si no coincideix amb el primer patró, ho intenta amb el següent *Patro2*, i així successivament. Si no coincideix amb cap dels patrons, el missatge es guarda per al seu posterior processament, i el procés espera al següent missatge.
- Els patrons i les guardes utilitzats en un comunicat de recepció tenen exactament la mateixa forma sintàctica i significat que els patrons i les guardes que usem quan definim una funció.

Processos

A la línia 2, creem un nou procés paral·lel. *spawn(Fun)* crea un procés paral·lel que avalua la funció, i que retorna el *Pid*, que es representa amb un <0.104.0>.

```
-module (area_server).
-export ([loop/0]).
loop () -> receive
        {cercle, R} ->io: format (" L'àrea del cercle és ~p~n", [3.1415926 * R * R]), loop();
        {quadrat, R} ->io: format (" L'àrea del quadrat és ~p~n", [R * R]), loop();
        {rectangle, X, Y} -> io: format (" L'àrea del rectangle és ~p~n" ,[X * Y]), loop();
        Other -> io: format (" No se l'àrea de ~p. ~n" ,[ Other ]), loop()
        end.
```

```
1> c(area_server).
{ok,area_server}
2> Pid = spawn (fun area_server : loop /0).
<0.104.0>
3> Pid!{rectangle,5,5}.
 L'àrea del rectangle és 25
{rectangle,5,5}
4> Pid!{cercle,5}.
 L'àrea del cercle és 78.539815
{cercle,5}
```

A la línia 3 s'envia un missatge al procés. Aquest missatge coincideix amb el tercer patró

Després d'haver rebut un missatge, el procés mostra l'àrea del rectangle. Finalment, s'imprimeix al shell {rectangle, 5, 5}. Això és perquè el valor de Pid! Missatge es defineix per ser un missatge. Si enviem al procés un missatge que no entén, genera un missatge d'error per *Other.*

Processos

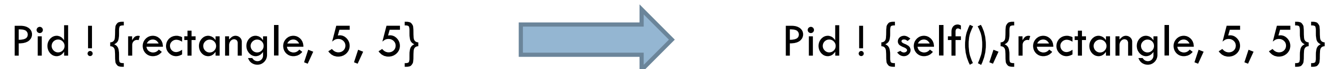

```
loop () -> receive
    {cercle, R} ->
        io: format (" L'àrea del cercle és ~p~n", [3.1415926*R*R]),
        loop();
    {quadrat, R} ->
        io: format (" L'àrea del quadrat és ~p~n", [R * R]),
        loop();
    {rectangle, X, Y} ->
        io: format (" L'àrea del rectangle és ~p~n" ,[X * Y]),
        loop();
    Other ->
        io: format (" No se l'àrea de ~p. ~n" ,[ Other ]),
        loop()
    end.
```

Processos

- El client i el servidor en una arquitectura client - servidor són processos separats, i la transmissió normal de missatges d'Erlang es la que s'utilitza per a la comunicació entre els. El client i el servidor es pot executar a la mateixa màquina o en dues màquines diferents.
- Les paraules client i servidor es refereix als papers que tenen aquests dos processos, el client sempre inicia una acció mitjançant l'enviament d'una sol·licitud al servidor. El servidor avalua la sol·licitud i envia una resposta al client.
- El programa anterior envia una sol·licitud a un procés i imprimeix el resultat. El que es vol fer ara, és enviar una resposta al procés que ha enviat la sol·licitud original. El problema és que no sabem a qui s'ha d'enviar la resposta. Per enviar una resposta, el client ha d'incloure un identificador a la que el servidor pot respondre.
- És com enviar una carta a algú, per obtenir una resposta s'ha d'afegir la seva adreça a la carta!

Processos

- El remitent ha d'incloure una adreça de resposta.



"self ()" és el PID del procés del client.

Per respondre a la sol·licitud, s'ha de canviar el codi que rep les peticions:

```
loop() -> receive
              {rectangle, X, Y} ->
                     io:format("L'àrea del rectangle és ~p~n" ,[X * Y]),
                     loop()
              ...
```

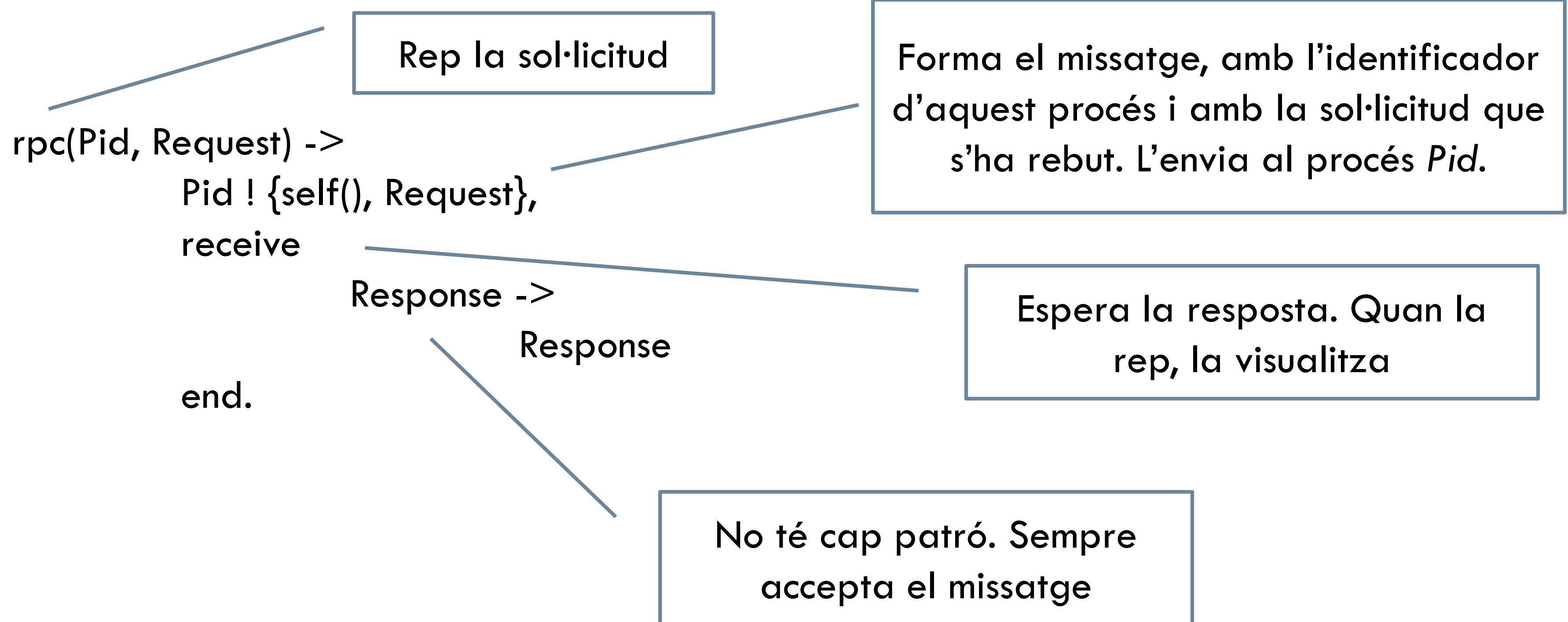
```
loop() -> receive
              {From, {rectangle, X, Y}} ->
                     From ! X* Y,
                     loop();
              ...
```

El procés que envia la sol·licitud inicial s'anomena un client. El procés que rep la sol·licitud i envia una resposta s'anomena un servidor.

Observeu la manera que envia el resultat del càlcul al procés identificat pel paràmetre *From*. Com que el client estableix aquest paràmetre en el seu propi procés d'identificació, rebrà el resultat.

Processos

Finalment, s'ha d'afegir una petita funció d'utilitat anomenada RPC (abreviatura de Remote Procedure Call) que encapsula el missatge i l'envia a un servidor i espera una resposta:



Pas de missatge síncron

Processos


```
-module(area_server1).
-export([loop/0, rpc/2]).

rpc(Pid, Request) ->
        Pid ! {self(), Request},
        receive
                Response ->
                        Response
        end.

loop() ->
        receive
                {From, {rectangle, X, Y}} -> From ! X * Y, loop();
                {From, {cercle, R}} -> From ! 3.14159 * R * R, loop();
                {From, Other} -> From ! {error,Other}, loop()
        end.
```

El procés rpc envia un missatge a loop

El procés loop envia un missatge a rpc amb la resposta

Visualitza la resposta

```
1> Pid = spawn(fun area_server1:loop/0).
```

Procés: loop

```
2> area_server1:rpc(Pid, {rectangle,6,8}).
48
```

```
3> area_server1:rpc(Pid, {cercle,6}).
113.097
```

```
4> area_server1:rpc(Pid, quadrat).
{error,quadrat}
```

Crida al procés rpc()

Processos

El mateix exemple una mica arreglat


```
-module(area_server2).
-export([loop/0, rpc/2]).


rpc(Pid, Peticio) ->
        Pid ! {self(), Peticio},
        receive
                Resposta ->
                        Fig=element(1,Peticio),
                        io:format("L'àrea del ~p és de ~p.~n", [Fig, Resposta])
        end.
loop() ->
        receive
                {From, {rectangle, X, Y}} -> From ! X * Y, loop();
                {From, {cercle, R}} -> From ! 3.14159 * R * R, loop();
                {From, Other} -> From ! {error,Other}, loop()
        end.
```

```
1> Pid = spawn(fun area_server2:loop/0).
<0.268.0>
2> area_server2:rpc(Pid, {rectangle,6,8}).
L'àrea del rectangle és de 48.
ok
3> area_server2:rpc(Pid, {cercle,6}).
L'àrea del cercle és de 113.09723999999999.
ok
```

Retorna l'element 1 de la tupla "Petició"

Processos

- Hi ha un petit problema amb aquest codi. A la funció *rpc*, s'envia una sol·licitud al servidor i espera una resposta. Però no espera una resposta del servidor, si no que espera qualsevol missatge. Si algun altre procés envia al client un missatge mentre està esperant una resposta del servidor, es pot mal interpretar el missatge com una resposta del servidor. Podem corregir això canviant la forma de la instrucció per rebre aquest:

```
loop() ->
    receive
        {From, ...} ->
            From ! {self(), ...}
            loop()
        ...
```

```
rpc(Pid, Request) ->
    Pid ! {self(), Request},
    receive
        {Pid, Response} ->
            Response
    end.
```

Quan s'entra a la funció *rpc*, el *Pid* està lligat a algun valor, de manera que en el patró {Pid, Resposta}, el Pid està lligat, i la resposta no està consolidat. Aquest patró coincideix amb un missatge que conté una tuple doble on el primer element és Pid. Tots els altres missatges es posen a la cua. (El receptor ofereix el que s'anomena recepció selectiva)

Processos


```
-module(area_server3).
-export([loop/0, rpc/2]).

rpc(Pid, Peticio) ->
        Pid ! {self(), Peticio},
        receive
                {Pid, Resposta} ->
                        Fig=element(1,Peticio),
                        io:format("L'àrea del ~p és de ~p.~n", [Fia, Resposta])
        end.

loop() ->
        receive
                {From, {rectangle, X, Y}} -> From ! {self(), X * Y}, loop();
                {From, {cercle, R}} -> From ! {self(), 3.14159 * R * R}, loop();
                {From, Other} -> From ! {self(), {error,Other}}, loop()
        end.
```

```
1> Pid = spawn(fun area_server2:loop/0).
<0.37.0>
3> area_server2:rpc(Pid, {cercle, 5}).
78.5397
```

Processos

- Hi ha una millora final que es pot fer. Podem amagar la *spawn* i la *rpc* dins del mòdul. Aquesta és una bona pràctica, perquè serem capaços de canviar els detalls interns del servidor sense haver de canviar el codi de client. Finalment, tenim el següent:

```
-module(area_server_final).
-export([start/0, area/2]).

start() -> spawn(fun loop/0).

area(Pid, What) -> rpc(Pid, What).

rpc(Pid, Request) ->
    Pid ! {self(), Request},
        receive
            {Pid, Response} -> Response
    end.
```

```
loop() ->
    receive
        {From, {rectangle, Width, Ht}} ->
            From ! {self(), Width * Ht},
            loop();
        {From, {circle, R}} ->
            From ! {self(), 3.14159 * R * R},
            loop();
        {From, Other} ->
            From ! {self(), {error,Other}},
            loop()
    end.
```

```
Pid = area_server_final:start().
area_server_final:area(Pid, {rectangle, 10, 8}).
```

Processos


```
-module ( area_server1 ).
-export ([loop/0]).
loop () -> receive
            {From, {cercle, R}} -> From! 3.1415926 * R * R, loop ();
            {From, {quadrat, R}} -> From! R * R, loop ();
            {From, {rectangle, X, Y}} -> From! X * Y, loop ();
            {From, Other} -> From ! {self(), {error,Other}}, loop()
            end.
```

```
1> Pid = spawn (fun area_server1 : loop /0).
<0.33.0>
2> Pid ! { self () , { cercle , 20}}.
{<0.31.0>,{cercle,20}}
3> receive V -> io: format (" Area : ~p~n",[V]) end .
 Area : 1256.63704
Ok
```

Processos


```
-module(area_server_final).
-export([start/0, area/2]).

start() -> spawn(fun loop/0).

area(Pid, What) ->  rpc(Pid, What).

rpc(Pid, Request) ->  Pid ! {self(), Request},
    receive
            {Pid, Response} ->  Response
    end.

loop() -> receive
            {From, {rectangle, Width, Ht}} ->  From ! {self(), Width * Ht}, loop();
            {From, {circle, R}} -> From !  {self(), 3.14159 * R * R}, loop();
            {From, Other} -> From ! {self(), {error,Other}}, loop()
    end.
```

```
1> c(area_server_final).
{ok,area_server_final}
2> Pid = area_server_final:start().
<0.52.0>
3> area_server_final:area(Pid, {rectangle,2,3}).
6
```

Processos


```
-module(calculadora).
-export([inici/0]).
inici () -> spawn(fun calcular/0).
calcular() -> receive
      {suma, X, Y} -> io:format ("~p + ~p = ~p ~n", [X, Y, X+Y]),  calcular();
      {resta, X, Y} ->  io:format ("~p - ~p = ~p ~n", [X, Y, X-Y]),  calcular();
      {multiplica, X, Y} -> io:format ("~p * ~p = ~p ~n", [X, Y, X*Y]), calcular();
      {divideix, X, Y} when Y > 0 -> io:format ("~p / ~p = ~p ~n", [X, Y, X/Y]), calcular();
      finalitzar -> void;
      Altre -> io:format("Error: comanda <~p> incorrecte~n", [Altre]),  calcular()
  end.
```

La funció *inici()* crea un nou procés amb la funció *calcular()*. Aquesta funció té com a missió realitzar els càlculs que se li passen per missatges.
Els patrons elegits són tuples en el qual el primer element és l'operador a realitza i la resta els valors a operar.
Sempre després d'operar es crida a la funció *calcular()* per permetre que se segueixin realitzant càlculs.

Processos

L'operació *divideix* està condicionada amb una guarda perquè no es puguin fer divisions per zero.
El patró *finalitzar* dóna per acabada la recepció de missatges. Deixant d'operar la calculadora.
El patró *Altre* serveix per cobrir tots aquells missatges que no estan permesos a mode d'error.

```
1> c(calculadora).
{ok,calculadora}
2> Calc = calculadora:inici().
<0.637.0>
3> Calc!{suma, 2, 4}.
2 + 4 = 6
{suma,2,4}
4> Calc!{resta, 6, 4}.
6 - 4 = 2
{resta,6,4}
5> Calc!{multiplica, 6, 4}.
6 * 4 = 24
```

```
{multiplica,6,4}
6> Calc!{divideix, 6, 4}.
6 / 4 = 1.5
{divideix,6,4}
7> Calc!{divideix, 6, 0}.
Error: comanda <{divideix,6,0}>
incorrecte
{divideix,6,0}
8> Calc!finalitzar.
Finalitzar
9> Calc!{suma, 2, 4}.
{suma,2,4}
```

- Quin és el rendiment? Si estem creant centenars o milers de processos Erlang, hem d'estar pagant algun tipus de penalització.
- Mesurem el temps que triga a generar un gran nombre de processos.

```
max(N) ->
   Max = erlang:system_info(process_limit),
   io:format("Nombre màxim de processos:~p~n",[Max]),
   statistics(runtime),
   statistics(wall_clock),
   L = for(1, N, fun() -> spawn(fun() -> wait() end) end),
   {_, Temps1} = statistics(runtime),
   {_, Temps2} = statistics(wall_clock),
   lists:foreach(fun(Pid) -> Pid ! die end, L),
   U1 = Temps1 * 1000 / N,
   U2 = Temps2 * 1000 / N,
   io:format("Temps de creació =~p (~p) microsegons~n", [U1, U2]).

wait() -> receive        die -> void  end.

for(N, N, F) -> [F()];
for(I, N, F) -> [F()|for(I+1, N, F)].
```

Quantitat de processos que pot executar.

Retorna el temps que ha passat des de l'últim cop que s'ha cridat. Suma del temps d'execució de tots els fils.

Mira el temps real d'execució (nº de cicles).

Servidor

```
1> processes:max(30000).
Nombre màxim de processos:262144
Temps de creació =5.2 (7.8) microsegons
```

Processos

Temps d'espera màxim

- Una recepció podria esperar eternament un missatge que mai arriba. Això pot passar per una sèrie de raons. Per exemple, hi pot haver un error lògic en el nostre programa, o el procés que ens anava a enviar un missatge es podria haver perdut abans d'enviar el missatge.
- Per evitar aquest problema, es pot afegir un temps d'espera de la recepció del missatge. Estableix un temps màxim que el procés esperarà per rebre un missatge.

```
receive
    Patro1 [when Guarda1] ->
        Expressions1;
    Patro2 [when Guarda2] ->
        Expressions2;
    ...
    PatroN [when GuardaN] ->
        ExpressionsN
    after Temps ->
        Expressions
    end
```

Sense el punt i coma (;)

Si no hi ha cap missatge que coincideixi amb els "patrons" abans de *Temps* en mil·lisegons, el procés deixarà d'esperar el missatge i avaluarà *Expressions*.

Processos

- Es pot implementar un temporitzador senzill utilitzant la recepció amb temps d'espera.
- La funció stimer1: start (Temps) avaluarà Fun (una funció sense arguments) després de *Temps* ms. Es retorna un identificador (un PID), que es pot utilitzar per cancel·lar el temporitzador.

```
start(Time) -> spawn(fun()
            -> timer(Time) end).

cancel(Pid) -> Pid ! cancel.
```

```
timer(Time) ->
    receive
        cancel -> io:format("S'ha cancel·lat~n")
        after Time -> io:format("He acabat~n")
    end.
```

```
1> Pid = stimer1:start(5000).
He acabat
2> Pid1 = stimer1:start(25000).
3> stimer1:cancel(Pid1).
cancel
```

Després de 5 segons

Inici d'un temporitzador i cancel·lació abans que el període del temporitzador ha expirat.

Mireu com està fet el mòdul stimer

Processos

Recepció selectiva

- La comanda per enviar missatges realment no envia un missatge a un procés. El que fa, és enviar un missatge a la bústia del procés, i la funció de rebre intenta eliminar el missatge de la bústia de correu.
- Cada procés en Erlang té associat una bústia. Quan s'envia un missatge al procés, es col·loca a la bústia. La bústia només s'examina quan el programa avalua una declaració de recepció:

```
receive
    Patro1 [when Guarda1] ->
        Expressions1;
    Patro2 [when Guarda1] ->
        Expressions1;
    ...
after
    Time ->
        ExpressionsTimeout
end
```

Processos

Recepció selectiva

- Quan entrem en la recepció d'un missatge, s'inicia un temporitzador (si hi ha una secció *after* a l'expressió).
- Agafa el primer missatge de la bústia de correu i el compara amb el Patró1, Patró2, i així successivament. Si la comparació té èxit, el missatge s'elimina de la bústia de correu, i s'avaluen els següents missatges.
- Si el primer missatge de la bústia no coincideix amb cap dels patrons de recepció s'elimina de la bústia i es posa en una "cua". Es fa el mateix amb el segon missatge de la bústia. Aquest procediment es repeteix fins que un missatge coincideix o fins que s'han examinat tots els missatges de la bústia de correu.
- Si cap dels missatges de la bústia coincideix, llavors el procés se suspèn i es reprograma per tornar-se a executar quan arribi un nou missatge a la bústia de correu. Nota: quan arriba un nou missatge, els missatges de la cua no es relliguen al nou temporitzador, només queda lligat el nou missatge.
- Tan aviat com s'hagi lligat amb un missatge, tots els missatges que s'han posat a la cua es tornen a introduir a la bústia en el mateix ordre en el que han arribat. Si hi ha un temporitzador activat, s'esborra.
- Si s'arriba al final de la temporització, s'avaluen les expressions *ExpressionsTimeout* i es desen els missatges guardats a la cua a la bústia en el mateix ordre en que han arribat.

- Si es vol enviar un missatge a un procés, s'ha de conèixer el seu PID. Això pot ser un inconvenient, ja que el PID s'ha d'enviar a tots els processos del sistema que volen comunicar-se amb aquest procés.
- Erlang té un mètode per a la publicació d'un identificador de procés de manera que qualsevol procés del sistema pot comunicar-se amb aquest procés. Tal procés s'anomena **processos registrats.**
- Hi ha quatre funcions especials per la gestió de processos registrats:

register(AnAtom, Pid)

Registra el Pid d'un procés a l'àtom AnAtom. El registre falla si AnAtom ja s'ha utilitzat per registrar un procés.

unregister(AnAtom)

Esborra el registre associat a l'àtom AnAtom.

Nota: Si un procés finalitza o mort s'esborra automàticament del registre.

whereis(AnAtom) -> Pid | undefined

Retorna el Pid associat a l'atom AnAtom. Si el procés no està registrat retorna un undefined.

registered() o regs()-> [AnAtom::atom()]

Retorna una llista de processos registrats al sistema (àtoms).

Processos registrats: exemple

A l'exemple de la calculadora, podem modificar la funció "inici"

```
inici () -> spawn(fun calcular/0).
Per:
inici () -> register(calc, spawn(fun calcular/0) ).
```

1> c(calculadora1).	{ok,calculadora1}
2> registered().	[Llista de processos]
3> whereis(calc).	undefined
4> calculadora1:inici().	true
5> whereis(calc).	<0.40.0>
6> registered().	[Llista de processos,calc]
7> calc!{suma,4,5}.	4 + 5 = 9 {suma,4,5}
8> calc!finalitzar.	Finalitzar
9> registered().	[Llista de processos]

Es pot veure com en principi el procés no està registrat, i quan fem la crida a calculadora1: inici/0 queda registrat a l'àtom calc. Podem utilitzar l'àtom per enviar els missatges i un cop acabada l'execució del procés s'extreu automàticament del registre.

Processos

Rellotges

Podem utilitzar els registres per crear un rellotge

```
start(Time, Fun) ->
    register(clock, spawn(fun() -> tick(Time, Fun) end)).

stop() -> clock ! stop.

tick(Time, Fun) ->
    receive
        stop ->
            void
        after Time ->
            Fun(),
            tick(Time, Fun)
    end.
```

```
11> c(clock).
{ok,clock}
12> clock:start(5000, fun() -> io:format("TICK
~p~n", [erlang:now()]) end).
true
TICK {1348,677022,63000}
TICK {1348,677027,79000}
TICK {1348,677032,94000}
TICK {1348,677037,110000}
TICK {1348,677042,125000}
13> clock:stop().
stop
```

Retorna el temps actual en us

Processos

Exemple clàssic: Ping Pong


```
-module(ping_pong).
-export([inici/0, inici/1, ping/2, pong/0]).

inici() ->    inici(10).

inici(Max) ->
    Pong = spawn(ping_pong, pong, []),
    Ping = spawn(ping_pong, ping, [Pong, Max]),
    Ping ! pong,
    ok.

 ping(Pong, Max) ->
    receive
        pong when Max > 0 ->
            io:format("Ping ~p~n", [Max]),
            Pong ! {self(), ping, Max},
            ping(Pong, Max -1);
        _ ->
            Pong ! finalitzar,
            io:format("Ping fet~n")
    end.
```

Ping envia un missatge a *Pong* i *Pong* respon.

```
pong() ->
    receive
        {Ping, ping, Max} ->
            io:format("Pong ~p~n", [Max]),
            Ping ! pong,
            pong();
        _ ->
            io:format("Pong fet~n")
    end.
```

La funció *inici/1* inicia els dos processos. S'utilitza la funció *spawn/3* per passar paràmetres a la funció, la seva sintaxi és *spawn (Mòdul, Funció, Llista_paràmetres)*. El procés *Pong* no té paràmetres. El procés *Ping* té com a paràmetres la referència al procés *Pong* i el nombre de vegades que es vol enviar el missatge.

Processos

Exemple clàssic: Ping Pong

- El procés *Ping* envia el missatge *ping* al procés *Pong*, mentre el comptador (*Max*) sigui més gran que zero. El missatge *ping* és una tupla que consta de la referència del procés *Ping* , l'àtom *ping* i el valor del comptador (*Max*).
- Cada vegada que *Ping* envia un missatge es decrementa el comptador. Quan el comptador arriba a zero envia un missatge per acabar al procés *Pong* (*Pong!finalitzar*).
- Quan el procés *Pong* rep el missatge enviat pel *Ping*, li respon amb un missatge *pong*. Si rep un altre missatge que no sigui *ping* finalitza el procés (no es torna a cridar a si mateix).

```
1> ping_pong:inici(3).
Ping 3
ok
Pong 3
Ping 2
Pong 2
Ping 1
Pong 1
Ping fet
Pong fet
```

Processos

Ping Pong modificat

Un procés *Pong* a de respondre a N processos *Ping*

```
inici() ->  inici(3, 3).

inici(Max, NumPing) ->
    Pong = spawn(ping_pong2, pong, [NumPing]),
    F = fun(I) ->
            Ping = spawn(ping_pong2, ping, [I, Pong, Max]),
            io:format("Inici del procés ~p~n",[I]),
            Ping ! pong
        end,
    lists:foreach ( F, lists:seq(1, NumPing) ).

ping(I, Pong, Max) ->
    receive
        pong when Max > 0 ->
            io:format("Procés Ping ~p ~n", [I]),
            Pong ! {self(), ping, I},
            ping(I, Pong, Max -1);
        _ ->
            Pong ! finalitzar,
            io:format("Fi Ping ~p~n", [I])
    end.
```

Aplica la funció F a cada element de la llista

```
-module(ping_pong2).
-export([inici/0, inici/2, ping/3, pong/1]).

pong() ->
    receive
        {Ping, ping, Max} ->
            io:format("Procés Pong ~p~n", [Max]),
            Ping ! pong,
            pong();
        _ ->
            io:format("Fi Pong fet~n")
    end.
```

Processos

Exemple clàssic: Ping Pong

- La funció *inici/2* crea un *Pong* que ha de respondre a tots els pings. I crea tots els *Pings* establerts en *NumPing*.
- El procés *Ping* rep un identificador *I* per visualitzar el nº del procés *Ping*. Funciona pràcticament igual que la versió anterior només s'afegeix l'identificador *I* en el missatge ping.
- En el *Pong*, per saber quan ha de finalitzar realment el procés s'ha afegit un paràmetre que és el nombre de pings generats. (només ha d'acabar quant tots els processos *Ping* han acabat)
- En conclusió, la comunicació entre processos pot ser bastant simple i amb poca feina es pot crear un sistema client/servidor. En aquests cas, el servidor és el procés *Pong*, que s'encarrega de despatxar totes les peticions dels seus clients, els processos *Ping's*.

```
1> ping_pong2:inici(2,2).
Inici del procés 1
Inici del procés 2
Procés Ping 1
Procés Ping 2
Procés Pong 1
ok
Procés Pong 2
Procés Ping 1
Procés Ping 2
Procés Pong 1
Procés Pong 2
Fi Ping 1
Fi Ping 2
Fi Pong
```

Índex

1. Introducció a Erlang
2. Programació concurrent
3. **Sistemes distribuïts**

Nodes

- Un node és una execució del ERS (Erlang Runtime System) que se li ha donat nom. El nom del node s'estableix al engegar l'Erlang amb els paràmetres -name i -sname (nom i nom curt respectivament).
- Un node és una unitat de treball que s'identifica unívocament per l'àtom nom@màquina (nom llarg) on nom és el nom curt i màquina és la màquina on s'està executant el node.
- És important, si volem treballar amb màquines remotes, que aquestes es puguin veure entre si.
- Una màquina pot tenir diversos nodes i el mateix nom curt pot estar utilitzat en diverses màquines.
- El paràmetre –name es sol utilitzar per nomenar nodes que es volen utilitzar de forma remota. -sname es sol utilitzar per nomenar nodes locals.

node1@nomMaquina node2@nomMaquina.upc.edu

Nodes

- Si arrenquem un node veurem la consola d'aquesta manera:

```
$ erl -sname node1
Erlang /OTP 21[erts-10.0]....
Eshell V10.0  (abort with ^G)
(node1@màquina)1>
```

- El *prompt* del Shell canvia indicant el node actual.
- Hi ha una funció que ens permet saber el node actual ***node*/0**. Si no hem creat el node la funció ens retornarà l'àtom nonode@nohost.

Connectant dos nodes del mateix domini (Amb Windows s'ha de desactivar el tallafocs)

En una consola podem arrencar un node (node1) i en una altre consola un altre (node2)

```
$ erl -sname node1 -setcookie clau
$ erl -sname node2 -setcookie clau
```

Per verificar la connexió:
(node1@màquina)1>net_adm:ping(node2@màquina).
pong

Si no reconeix el node, respon amb un:
pang

Nodes

- La connexió del node1 al node2 es pot fer amb les comandes JCL (^G)

```
(node1@maquina1)4> ^G
User switch command
--> j
   1* {shell,start,[init]}
 --> r node2@maquina2
 --> j
   1  {shell,start,[init]}
   2* {node2@maquina2,shell,start,[]}
 --> c
Eshell V5.7.4  (abort with ^G)
(node2@maquina2)1>
```

Es pot anar d'un treball a un altre i finalitzar el treball amb el node2

```
(node2@maquina2)1> nodes().
[node1@maquina1]
(node2@maquina2)2>
User switch command
 --> j
   1  {shell,start,[init]}
   2* {node2@maquina2,shell,start,[]}
 --> c 1
(node1@maquina1)4>
User switch command
 --> c 2
(node2@maquina2)2>
User switch command
 --> k 2
 --> c
Unknown job
 --> j
   1  {shell,start,[init]}
 --> c 1
(node1@maquina1)4>
```

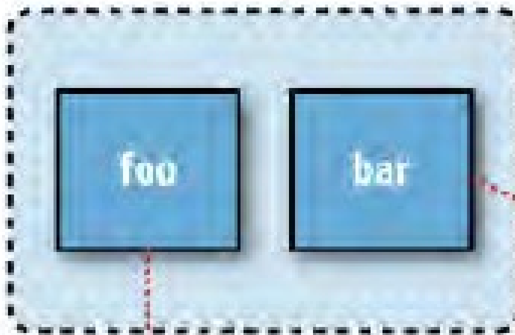
Nodes

- Un node també es pot crear amb: net_kernel

```
1> erlang:is_alive().
false
2> net_kernel:start([n1, shortnames]).
{ok,<0.38.0>}
(n1@maquina)2> erlang:is_alive().
true
(n1@maquina)3> node().
n1@maquina
(n1@maquina)8> net_kernel:stop().
ok
9>
```

Sistemes distribuïts

STC.kent.ac.uk



dist.erl

```
-module(dist).
-export([t/1]).
t(From) -> From ! node().
```

```
erl -sname foo
...
(foo@STC)1>spawn('bar@STC', dist, t, [self()]).
<5734.42.0>
(foo@STC)2> flush().
Shell got 'bar@STC'
ok
```

```
erl -sname bar
...
(bar@STC)1>
```

Sistemes distribuïts

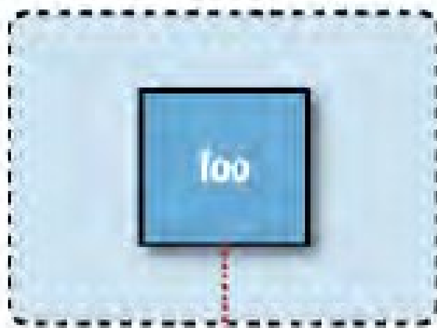
Terminal 1

```
1 > net_kernel:start([p1, shortnames]).
{ok,<0.34.0>}
(p1@nom)3> spawn('p2@nom', dist, t, [self()]).
<6602.57.0>
(p1@nom)7> receive V -> V end.
p2@nom
```

Terminal 2

```
1> net_kernel:start([p2, shortnames]).
{ok,<0.42.0>}
(p2@nom)2> c(dist).
{ok,dist}
(p2@nom)3>
```

STC.kent.ac.uk



dist.erl

```
-module(dist).
-export([t/1]).
t(From) -> From ! node().
```

FCC.erlang-consulting.com

```
erl -sname foo -setcookie cake
...
(foo@STC)1> spawn('bar@FCC', dist, t, [self()]).
<5734.42.0>
(foo@STC)2> flush().
Shell got 'bar@FCC'
ok
```

```
erl -sname bar -setcookie cake
...
(bar@FCC)1>
```

$ erl –sname foo –setcookie clau

Nom curt

```
1> net_kernel:start([foo, shortnames]). {ok,<0.33.0>}
(foo@Maquina)2> erlang:set_cookie(node(), 'cake').  true
(foo@Maquina)3> erlang:get_cookie(). cake
(foo@Maquina)4> spawn('bar@maquina', dist, t, [self()]). <0.45.0>
(foo@Maquina)4> net_kernel:stop(). ok
```

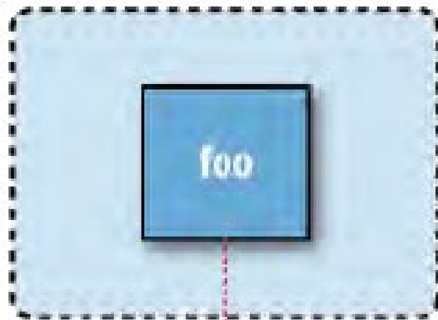
$ erl -name foo –setcookie clau

nom llarg

```
5 > net_kernel:start(['foo@adreça IP', longnames]). {ok,<0.47.0>}
(foo@adreça IP)6> erlang:set_cookie(node(), 'cake').  true
 (foo@adreça IP)7> net_adm:ping('bar@192.168.1.35').  pong
(foo@adreça IP)7> spawn('bar@192.168.1.35', dist, t, [self()]). Envio 'bar@adreça IP’
<9123.44.0>
(foo@adreça IP)8> flush().  Shell got 'bar@adreça IP'
ok
```

Condició: L’adreça IP ha de ser pública.
Si no ho és, s’ha d’obrir el port que utilitza al servidor.

STC.kent.ac.uk

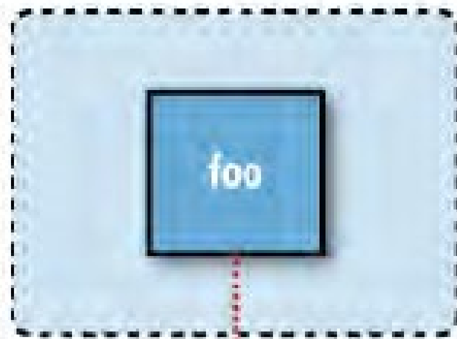


FCC.erlang-consulting.com

```
erl -sname foo -setcookie fish
...
(foo@STC)1> net_adm:ping('bar@FCC').
pang
(foo@STC)2> erlang:set_cookie(node(), cake).
true
(foo@STC)3> net_adm:ping('bar@FCC').
pong
```

```
erl -sname bar -setcookie cake
...
(bar@FCC)1>
=ERROR REPORT====30-Aug-2008::14:15:38===
**Connection attempt from disallowed node
'foo@STC'**
(bar@FCC)1>
```

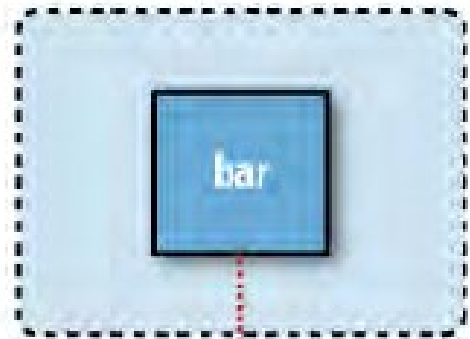
STC.kent.ac.uk



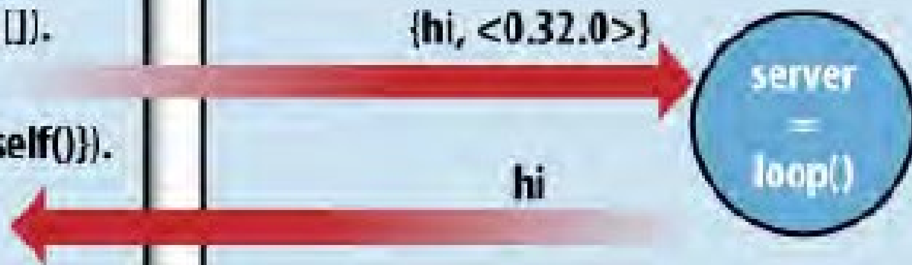
dist2.erl

```
-module(dist2).
-export([s/0]).
s() -> register(server, self()), loop().
loop() -> receive {M, Pid}
                 -> Pid ! M end,
        loop().
```

FCC.erlang-consulting.com




```
erl -sname foo -setcookie cake
...
(foo@STC)1> spawn('bar@FCC', dist2, s, []).
<4824.44.0>
(foo@STC)2> {server, 'bar@FCC'} ! {hi, self()}.
{hi, <0.32.0>}
(foo@ STC)3> flush().
Shell got hi
```



```
-module(dist2).
-export([s/0]).

s() ->
        register(server, self()),
        loop().

loop() ->
        receive {M, Pid} ->
                Pid!M
        end,
        loop().
```